Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Alvaro Garcia de la Villa

# Distributed Denial of Service Attacks defenses and OpenFlow:

## Implementing denial-of-service defense mechanisms with software defined networking

Final Project
Espoo, September 12, 2014

Supervisors: Professor Tuomas Aura
Advisor: Aapo Kalliola M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
FINAL PROJECT

| | | | |
|---|---|---|---|
| **Author:** | Alvaro Garcia de la Villa | | |
| **Title:** | Distributed Denial of Service Attacks defenses and OpenFlow: Implementing denial-of-service defense mechanisms with software defined networking | | |
| **Date:** | September 12, 2014 | **Pages:** | 65 |
| **Major:** | Computer Science and Engineering | **Code:** | T |
| **Supervisors:** | Professor Tuomas Aura | | |
| **Advisor:** | Aapo Kalliola M.Sc. (Tech.) | | |

DoS attacks are becoming more common, as well as emerging new technologies, to separate the control plane and the data plane from network devices (SDN). Throughout this survey we will investigate how OpenFlow can help prevent and locate these kind of attacks. We will study the different DoS attacks and the current mitigation techniques. We will discuss as well, which of these techniques could be improved with OpenFlow and how to develop them. In the end, we will implement and test some identified mitigation techniques and we will study their behaviour.

| | |
|---|---|
| **Keywords:** | DoS, DDoS, SDN, OpenFlow |
| **Language:** | English |

# Acknowledgements

I would like to express my gratitude to my instructor Aapo Kaliola for the useful comments, remarks and engagement through the learning process of this final project. Furthermore I would like to thank Tuomas Aura for helping me in the final stretch.

Also, I will be always grateful to Finland for hosting me on it and all the finnish people which I will carry always in my heart. Thank you very much also to Aalto University, where I have learnt how to work and live with people all over the world. I would never forget my country, Spain, where I have grew up and learnt all I know.

But above all else, I dedicate this thesis to my parents, they are the people to whom I owe for making me into the person I am today. I will eternally be grateful.

Espoo, September 12, 2014

Alvaro Garcia de la Villa

# Abbreviations and Acronyms

| | |
|---|---|
| IP | Internet Protocol |
| TCP | Transmission Control Protocol |
| UDP | User Data Protocol |
| ICMP | Internet Control Message Protocol |
| VLAN | Virtual Local Area Network |
| MPLS | Multiprotocol Label Switching |
| TLS | Transport Layer Security |
| DoS | Denial of Service |
| DDoS | Distributed Denial of Service |
| ISP | Internet Service Provider |
| SDN | Software Defined Networking |
| SYN | synchronization |

# Contents

# List of Figures

# Chapter 1

# Introduction

With the rapid growth of the Internet over the last two decades, the number of attacks on the Internet services has increased rapidly. One such example involves disrupting the service provided by a network or server, either by crashing the systems, sending some packets that exploit a software vulnerability or by sending a large amount of useless traffic to collapse the resources of the service. This kind of attack is known as a Denial of Service (DoS) attack, or a Distributed Denial of Service attack if it is launched by multiple hosts.

The design principles of the Internet facilitate these kind of attacks [25]:

*Resource sharing*: in IP networks, due to the packet-switched service, users share all the resources, and one user's service can be disturbed by other user's behaviour, so bandwidth attacks can disrupt services for legitimate users.

*Simple Core and Complex Edge*: One of the principles of the Internet is that the core network should be simple and all complexity is pushed into the end hosts. That means that the core of the network is not able to integrate complex applications, such as those concerning authentication or security. Due to this simplicity, when an attacker sends packets to the network and the target receives them, it is almost impossible to verify the true origin of the packets.

*Fast Core Networks and Slow Edge Networks*: The core network needs to have a high capacity due to the heavy traffic that it has to route from many sources to many destinations. In contrast, an edge network needs less capacity because it only needs to support its end users. The consequence is that traffic from the high-capacity core can crush the slow-capacity edge.

A large number of different DoS and DDoS attacks have arisen to take advantage of these design principles and their vulnerabilities and, as a result,

a widely variety of defense mechanisms have risen in order to defeat these attacks.

In the current network architecture, network devices (particularly routers) are bundled with a specialized control plane and various features. This vertical integration essentially binds whatever software and features that are shipped with those particular devices. Software Defined Networking (SDN) effectively breaks these pieces apart.

*SDN* is a type of network architecture that separates the network data plane (network devices that forward traffic) from the control plane (software logic that ultimately controls how traffic is flowing through the network). *OpenFlow* [22] is a standard interface defined between the control and forwarding layers of an SDN structure.

One of the reasons to separate the control plane and the data plane is that the control software of the network can evolve independently from the hardware.

A second reason is that the separation allows the network to be controlled from a single high-level software program. There are a large number of *OpenFlow* controllers, depending on the programming language, difficulty, goal, etc. In this project, we will develop a controller with POX.

The goal of this thesis is to analyse how SDN might help us in improving the current DDoS defense mechanisms. Throughout this project, we will review the main DDoS defense and attack mechanisms and we will go through existing algorithms. We will also explain how we can improve them with *OpenFlow*. We will test these algorithms in virtual scenarios with Mininet.

This thesis is structured as follows: In the next chapter (see 2) we will explain the background of this survey. We will talk about the current situation of DDoS attacks and defenses and how OpenFlow works as well as its structure. Chapter 3 shows the current situation of *firewalls* and how they can help defeating DoS and DDoS. In this chapter there is also a comparison between *firewalls* and *SDN*, highlighting the advantages and disadvantages of both. Chapter 4 explains the *TCP SYN Flooding* Protocol Attack as well as its methods of attacks and defense mechanisms currently in the market. The algorithm developed to defeat TCP SYN Flooding attacks and the scenario where it is tested will be shown in Chapter see 5. Chapter 6 shows how defense mechanisms react against a *TCP SYN Flooding* attack and all possible information about its behaviour. Finally, the conclusions are shown in Chapter 7.

# Chapter 2

# Background

## 2.1 DoS attack and defense mechanisms

A denial-of-service (DoS) attack is characterised by an explicit attempt by attackers to prevent the legitimate users of a service from using the service [5] provided by a network or server. It can be launched in several ways, but this project is focussed in two of them [25]. The first aims at crashing the system by sending crafted packets that exploit a software vulnerability in the target system. The other way is by sending massive volumes of useless traffic to overwhelm and occupy the resources that could service legitimate traffic.

### 2.1.1 DoS and DDoS Attacks

The volume of traffic for the attack must be large enough to consume the target's resources. In order to deny services and accomplish more complicated attack detection, the attack is carried out through multiple sources. This variant of DoS attack is known as **DDoS (Distributed Denial of Service) attack**.

A typical DDoS attack contains three main elements as shown in Figure 2.1. First of all, the *attacker* selects a set of vulnerable systems (*zombies*) and sets up attack systems in them. Once the attack mechanisms are installed, the *attacker* can launch *attack commands* to the *zombies* through a secure channel to carry out the DoS attack on the *victim*. The complexity of the attack increases due to the *zombies* modifying the packets, commonly spoofing the source. As a consequence, it becomes even more difficult to trace the origin of the attack.

*Zombie systems*, also known as *bots* and the structure of elements the attacker can launch to attack systems over them, and carry out joint attacks,

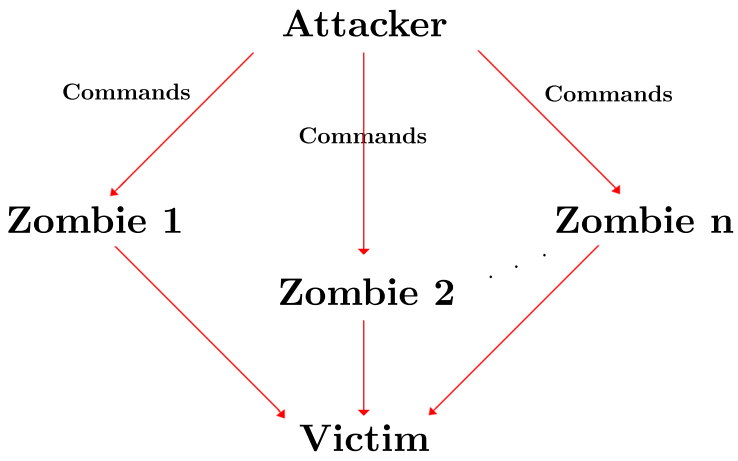



Figure 2.1: DDoS Attack Structure

is commonly known as *botnet*. An important feature of *botnets* is the ability to update software from the *attacker* through the security channel between the *attacker* and the *bots*.

### 2.1.2 Attack Taxonomy

In order to devise a taxonomy of DDoS attacks, we have to take into account some features of the attacks, as well as the means used to prepare and perform the attack, the characteristics of the attack itself and the selection and the effects upon the victim. In this survey, we will focus on selected attacks depending on the victim type, classifying them as Protocol Attacks (see 2.1.2.1), Bandwidth Attacks (see 2.1.2.2) or Logic Attacks (see 2.1.2.3). There are already some other taxonomies to explain all aspects in greater detail [11, 18, 33]

#### 2.1.2.1 Protocol Attacks

The attacker continuously sends packets to the server at a particular rate to take advantage of the inherent design of common network protocols. In other words, these attacks try to exploit the weaknesses of the system, considering the expected behaviour of protocols such as TCP, UDP, and ICMP.

*SYN Flooding Attacks* flood the server, by sending *SYN* packets that consume its resources and fill up the backlog (This survey is focused on this type of attack, which is further explained in Chapter 4). A *UDP Flood Attack* is another protocol attack which aims at bringing down the server by sending UDP packets to a random port in the target. For a large number of UDP packets, the target will be forced to send back ICMP packets, but to a unreachable destination [9]. Other examples are *Smurf Attacks* [1] and *ICMP Attacks*.

#### 2.1.2.2 Bandwidth Attacks

High-data-volume attacks can consume all available bandwidth between an ISP and a target. The ISP networks need to have a high capacity due to the heavy traffic that they have to route from many resources to many destinations. The connections between the ISP and the victim usually have less capacity than the ones inside the ISP, so when high volumes of traffic coming from the ISP go through these connections, the links fill up and legitimate traffic slows down. An attacker can consume bandwidth by transmitting any traffic to all the network connection [14]. For example, high volumes of simple ICMP packets to consume the bandwidth [3].

#### 2.1.2.3 Logic Attacks

In Logic or software attacks, a small number of malformed packets exploit known specific software bugs in the operating system or in an application of the target system. This can potentially disable the victim's machine with one or multiples packets. These attacks are relatively easy to avoid either through the installation of software that eliminates vulnerabilities [35] or by adding specialized filter rules to filter out malformed packets [19].

In *ping of death* attack, the attacker sends a *ping* message with the packet size over the limit (656536 octets) so that it is allowed to be retransmitted over the Internet. Other examples are *land attacks, Teardrop Attack* [9].

### 2.1.3 Defense Classification

DDoS defense mechanisms have become one of the most important challenges in network security. Therefore, a large number of defense classifications and taxonomies have emerged [18, 33]. In this thesis, we will present four broad categories. The purpose of this categorization has been to highlight the main features of each category of defense.

*Attack Prevention*: it aims at stopping the attacks before they can reach their target (see 2.1.3.1); *Attack Detection*: it aims at detecting the attack when it occurs (see 2.1.3.2); *Attack Source Identification*: it aims at locating the source of the attack (see 2.1.3.3); and *Attack Reaction*: aims at eliminating or reducing the effects of the attack (see 2.1.3.4).

#### 2.1.3.1 Attack Prevention

Its objective is to stop attacks before they actually cause damage. This type of category tries to deny traffic that can be recognized as malicious, based on known patterns. The best place to allocate these mechanisms is in the edge routers and hosts, which implies fixing all the vulnerabilities of all Internet hosts that can be misused for an attack. Some useful approaches to prevent DDoS attack against a target machine are:

**Filtering** This measure implies installing ingress and egress packet filters on all the routers. In order to protect the target from attacks arriving to the network and prevent the network itself from being a potential attacker, *filtering* all the packets entering and leaving the network might be a good option.

**Firewall** Before an attack is carried out, a *firewall* might be useful to filter out traffic according to the protocol, ports or incoming IP addresses. But, the problem is that firewalls cannot distinguish between an attack and legitimate traffic, and denying all traffic for a specific port or protocol is not suitable. Only in those attacks in which the signature patterns are known, may these patterns be avoided. However, a insignificant variation or new attacks can make the attack go undetected.

**Protocol Security** addresses the problem of protocol design weaknesses in order to prevent Protocol Attacks (see 2.1.2.1) such as a TCP SYN Attack, malformed packets, UDP Flooding, etc. [6]

#### 2.1.3.2 Attack Detection

Once the attack is in process, an *attack detection mechanism* must recognize if it is actually an attack or just legitimate traffic. Also, in an attack situation, legitimate traffic must flow without being misclassified and disrupted. When some traffic is confused with attack traffic while it is not, it is called *false positive*. *False positives* are those considered as legitimate traffic but that they are actually attacks. [4]

An effective attack detection mechanism must keep the balance between false positives and false negatives. There are basically two kinds of detection schemes [13, 35] :

**Pattern Detection** An attack can constantly be detected by comparing incoming traffic with known attacks signatures stored in a database. These patterns are constructed by network security experts based on previous attacks. If the attack matches the database, this mechanism becomes very efficient with almost no false positives. Problems arise when there are new attacks or slight variants that can dodge the defense. SNORT [28] and Bro [23] are two commonly used pattern detection approaches.

**Anomaly Based Detection** It identifies malicious activity in a network by detecting anomalous network traffic patterns. Some network analysis behaviour such as detecting the attacks based on the size of the packet, since those being too short violate specific application layers protocols. *Rate-based detection* is also an important network analysis. It perceives changes in the traffic flow, detecting floods by using a time-based model of normal traffic volumes.

The parameters on which the defense mechanism is based to detect the anomaly can be [13]: *Standard*, they relay on protocol standards-for example, an attack detection can detect half-open TCP connections-and *Trained*, which generates allowed threshold values normal conditions based on the system's behaviour under normal conditions.

#### 2.1.3.3 Attack Source Identification

Once an attack is detected, the best response is to block the attack traffic at its source. It aims at locating the attack sources regardless of whether the source address field in each packet contains correct or erroneous information. Once the attack detection phase is over, the IP attack traffic should be traced back to its source. This is taken care of in phase [25].

Unfortunately, it is not easy to track IP traffic down to its source. This is due to two aspects of the IP protocol. The first is the ease with which IP source addresses can be forged. The second is the stateless nature of IP routing, where routers normally know only the next step for forwarding a packet, instead of the complete end-to-end route taken by each packet.

#### 2.1.3.4 Attack Reaction

Attack reaction tries to eliminate the effects of an attack and filter the attack traffic without disturbing legitimate traffic. The reaction to the attack must minimize the damage caused by the attack by developing a reaction scheme while the attack is in progress.

**Filtering** Dropping the traffic considered as unwanted or malicious is an effective way to prevent a DDoS attack. The problem is that some attacks use well-formed packets and legitimate requests to servers, making them non-filterable. There is also the risk of accidentally denying service to legitimate traffic. However, it is an efficient mechanism against spoofed IP packets.

Dropping spoofed incoming packets by ingress filtering [8], identifying and dropping packets based on the change of the time-window-size, saving proved previously legitimate IPs [24], are some of the attack reaction mechanisms based on filtering.

**Rate Limiting** The rate of malicious traffic packets is reduced with this technique when there is a high number of false positives and traffic has been identified as malicious by the detection mechanisms.

*Max-Min Fair Share* sets up maximum and minimum thresholds by the routers fixed by the servers. *Level-K* controls the traffic admission rates of the routers; $k$ hops away the victim using a max-min fairness approach [37].

## 2.2 OpenFlow (SDN)

The explosion of mobile devices, server virtualization, security problems and the advent of cloud services are among the reasons for the networking industry to begin to question traditional network architecture. *OpenFlow* is intended to solve the problem of assigning resources to users in an easy way by giving them the control plane of the network without disturbing traffic flows.

In traditional routers and switches, both the control plane (high level routing decisions) and the data plane (packet forwarding) are embedded in the same device. An *OpenFlow Switch* separates these two functions (Figure 2.2). The data plane function still resides on the switch, while the control plane is moved to a separate device called the *Controller* (see 2.2.5) that manages the switch and communicates with each device over the *Secure Channel* (see 2.2.1.2), via the *OpenFlow Protocol* (see 2.2.2).

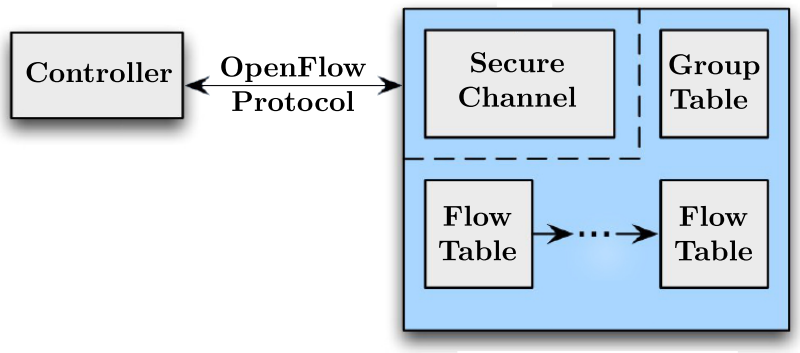



Figure 2.2: OpenFlow Switch Components

The switch contains *flow tables* 2.2.1.1, which are updated by means of *OpenFlow Protocol* by adding, updating and deleting *OpenFlow Switch flow entries*. When the traffic flow gets to the switch, it checks if the packets match the flow table (see 2.2.3); if so, the action defined at the flow entry is executed. Otherwise, the packet is either sent to the *Controller* or dropped.

Throughout this section, we will explain in detail the main parts of the *OpenFlow Switch*, as well as the *Controller* and how they work together. The first version of the *OpenFlow* (1.1) protocol was released on 2011, one year later, in February 2012, the ONF approved and published version 1.2. Presently, the current version of the protocol and the one that will be used in this project is 1.4 [34].

## 2.2.1 Switch Components

### 2.2.1.1 Flow Table

A flow table consists of several flow entries (see table 2.2.1.1), and each flow table entry contains:

- **Match Fields**: This consists of fields which match against the packets and tell the switch how a flow should be processed. The match fields (see table 2.2.1.1) are:
  - **Ingress Port**: It matches against the switch port number at which the packet arrives. This may be a physical or a switch-defined virtual port.

  - **Ether source**: It matches against the Ethernet source address.
  - **Ether dst**: It matches against the Ethernet destination address.
  - **Ether type**: Ethernet type of the *OpenFLow* packet payload. It differentiates between both standard Ethernet and 802.3.
  - **VLAN id**: It matches against the VLAN ID.
  - **VLAN priority**: It defines the VLAN ID priority.
  - **IP src**: It matches against the IP source address (IPv4 and IPv6). It can use subnet mask or arbitrary bitmask.
  - **IP dst**: It matches against the IP destination address (IPv4 and IPv6). It can use subnet mask or arbitrary bitmask.
  - **IP proto**: It defines TCP, UDP or ICMP protocol packets.
  - **IP ToS bits**: 8 bits value to specify the IP type of service.
  - **src port**: It matches against the source port machine the packet was sent from.
  - **dst port**: It matches against the destination port machine the packet is sent to.

- **Priority**: Integer number to define what priority should be followed in the flow entries. The lower the number, the higher the priority. The first flow entries defined have priority over the later ones.

- **Counters**: *Counters* are updated when packets are matched. They are maintained for each flow table, flow entry, port, queue, group, meter and meter band. A switch is not required to support all counters.

- **Instructions**: When a packet matches against a flow entry, a function is executed. In this field, the actions to be executed for those packets are defined. Some of these actions are: *Output*, forwarding a packet to a specified OpenFlow port (ALL, CONTROLLER, TABLE, INPUT and OUTPUT PORT number, FLOOD); *Drop*, it drops match packets...

- **Timeouts**: The maximum amount of time or idle time before the flow is expired by the switch. This field will be better explained in section 2.2.4.

- **Cookies**: They may be used by the controller to filter flow statistics, flow modification and flow deletion. These are not used when processing packets.

| Match Fields | Priority | Counters | Instructions | Timeouts | Cookie |
|---|---|---|---|---|---|

Table 2.1: Main Components of a flow entry

| Ingress Port | Ether source | Ether dst | Ether type | VLAN id | VLAN priority | IP src | IP dst | IP proto | IP ToS bits | src port | dst port |
|---|---|---|---|---|---|---|---|---|---|---|---|

Table 2.2: Main components of the Match Field

#### 2.2.1.2 Secure Channel

The message exchange between the *Controller* and the *OpenFlow Switch* is carried out over the *Secure Channel*. One controller can manage multiple secure channels into the topology, one per *OpenFlow Switch*.

The *Secure Channel* is usually instantiated as a single network connection between the switch and the controller, using TLS or plain TCP. Alternatively, the *OpenFlow Channel* may be composed of multiple network connections to exploit parallelism. The only requirement is that it should provide TCP/IP connectivity.

### 2.2.2 OpenFLow Protocol

*OpenFlow Protocol* is the protocol used for communication between the *Controller* and the *OpenFlow Switch*. It is built over the TCP/IP architecture and there are three different types of messages:

**Controller-to-Switch messages** The *Controller* sends messages to the switch in order to either perform actions or request information from the switch. These messages may or may not require a response from the *OpenFlow Switch*. The *Controller* may request the identity and the basic capabilities of the switches; it is able to send messages to manage the state of the switches in order to modify, update and add flow entries in the *OpenFlow Switch* flow tables; it can request information and statistics of the flow entries; it may send the packets directly to an specific switch port, which will be received via packet-in messages etc.

**Asynchronous messages** The *Controller* may receive messages from the switch with no need for the controller to ask for them. When a packet arrives at the switch and matches a flow entry with the instructions

set up by sending the packets to the *Controller*, the *Controller* receives an asynchronous message. The *Controller* may also receive the switch status changes.

The three main asynchronous messages types are: *Packet-in*, it transfers the control of a packet to the *Controller*. *Flow-Removed*, when a flow entry is removed, the switch informs the *Controller*. *Port-status*, it informs the controller of a change on a port.

**Symmetric messages** Symmetric messages are sent without request in a concurrent way and in both directions. The OpenFlow protocol uses symmetric messages when the connection starts (*Hello* messages); to verify the viability of a controller-switch connection (*Echo* and *Echo reply* messages); and for error messages on the other side of the connection.

*OpenFlow Protocol* fields are shown in figure A.3.

### 2.2.3 Matching

Figure 2.3 shows the functions that OpenFlow Switch performs when it receives a packet.

First of all, the switch starts by accomplishing a table lookup in the first flow table. Packet match fields used in the table lookup depend on the the flow table (see 2.2.1.1) match fields. A packet matches a flow table if the values in the packets match against the values defined in the fields of the flow table entry. When a packet matches against a flow entry, the action carried out is the one with the highest priority as defined in the field *Instructions* of the flow entries *(Priority field)*. Then, the *counters* associated to the flow entries selected must be updated.

Every flow table must support a table-miss flow entry in order to process table misses. The table-miss flow entry specifies how to process packets unmatched by other flow entries in the flow table, and it may send packets to the controller, drop packets or direct packets to the subsequent table.

### 2.2.4 Flow Removal

The size of the flow tables is not unlimited, so it is interesting to remove flow entries once they are not useful anymore. This function can be performed by either a request sent by the *Controller* or via the flow expiry mechanism.

The switch **flow expiry** is run by the switch and it is based on the the flow entries configuration, specifically the *Timeouts* field. There are two different

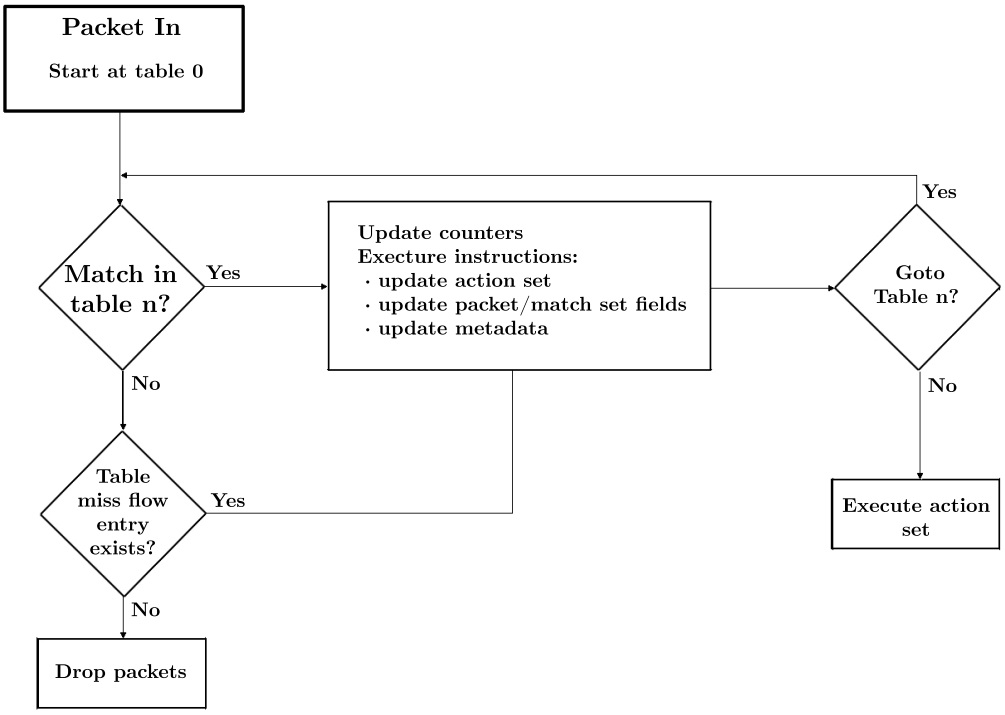



Figure 2.3: Flowchart of Packet Process

types of timeouts, **hard-timeout** and **idle-timeout**. **Hard-timeout** is the time (in seconds) a flow entry remains in the flow tables of the switch once the rule is set up until the timeout is over. **Idle-timeout** is the time (in seconds) a flow entry remains in the flow table of the switch since the rule is set up until the timeout is over, but each time a packet matches against the flow entry, the counter starts again. If no packets arrive before the timeout value, the flow entry is removed from the flow table, freeing up memory of the switch.

### 2.2.5 Controller

The *Controller* computes flow-entries based on the program or the protocol running on it. It also takes care of adding or deleting flow-entries from the flow table present in the *OpenFlow* switches. A Controller can either be a simple PC or a dedicated sophisticated machine for handling complex tasks and having multiple accounts for different researches to experiment for their

research projects.

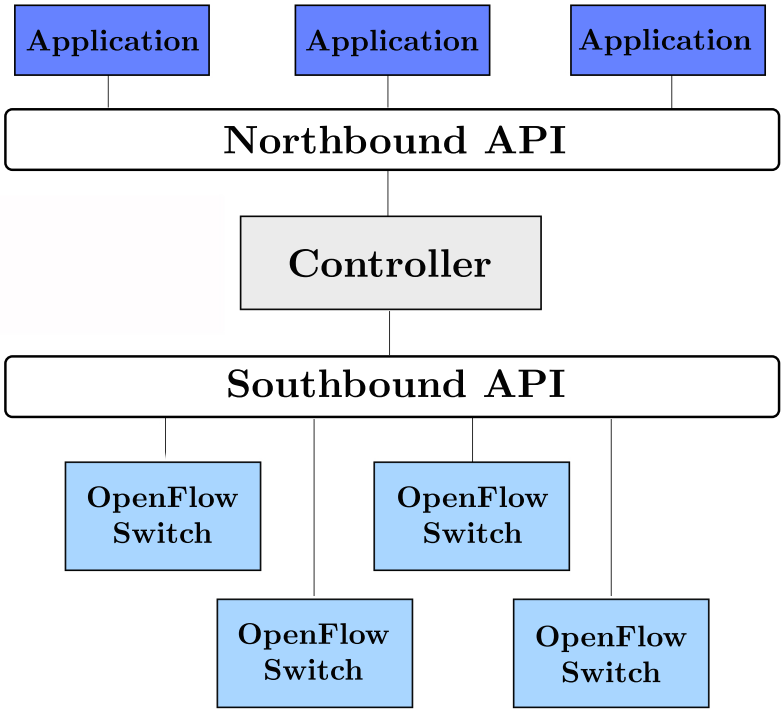



Figure 2.4: SDN API Structure

The natural question arising when using a central controller has to do with scalability, reliability and performance of using a single controller. These questions have been answered to some extent in one of the previous prototypes, in which a low-cost desktop PC could process around ten thousand flows per second. This is good enough for a large college campus. Even scalability and reliability could be accounted for by having multiple separate controllers, each one of them working on a stateless principle.

Figure 2.4 helps to understand the concepts **southbound** and **northbound**. A **southbound** interface allows a particular network component to communicate with a lower-level component, in this case the OpenFlow Switchs. On the other hand, a **northbound** interface is an interface that allows a particular component of a network to communicate with a higher-level component, in this case, the applications.

In the SDN architecture, which is studied in this project, the *southbound* interface might be compared with the *OpenFlow Protocol* 2.2.2. SDN controllers expose *northbound APIs*, allowing you to deploy a wide range of off-the-shelf and custom-built network applications.

Due to the complexity of developing the *OpenFlow Controller*, a wide variety of different SDN controllers compatible with *OpenFlow Protocol* have emerged to help the developer abstract to low-level programming. Table 2.2.5 shows some of the most popular SDN controllers compatible with *OpenFlow Protocol*.

| | **NOX** | **POX** | **Ryu** | **Floodlight** |
|---|---|---|---|---|
| **Language** | C++ | Python | Python | Java |
| **Performance** | Fast | Slow | Slow | Fast |
| **Learning Curve** | Moderate | Easy | Moderate | Steep |

Table 2.3: Most common SDN Controllers

#### 2.2.5.1 POX

POX controller [20] is a platform for the rapid development and prototyping of network control software using Python. It is based on NOX [21], a Network Operating System which provides a central and uniform programmatic interface to the whole of the OpenFlow network.

Like an Operating System which provides the ability to read, edit and write various media, POX provides the ability to view and control the network. Applications written on top of POX *(APIs Northbound)* would perform the actual network management tasks, while POX provides the following two major functionalities:

- Centralized Programming Model, where the programs would be written as if the entire network was present on a single machine.
- Programs are written in terms of a high level of abstraction.

# Chapter 3

# Firewall Security System

A Firewall is a network security system that is used to control the flow of ingress and egress traffic, usually between a more secure local-area network (LAN) and a less secure wide-area network (WAN). The system analyses data packets for parameters like L2/L3 headers (i.e., MAC and IP address) or performs deep packet inspection (DPI) for higher layer parameters to filter network traffic.

## 3.1 Types of Firewall

Basically, there are two different types of firewall, depending in which layer of the OSI (Open Systems Interconnect) model [36] the firewall is set up.

Firewalls can be divided between **Network Layer firewalls** (see 3.1.1) and **Application Layer firewalls** (see 3.1.2). The important thing to recognize is that he lower the level of the forwarding mechanism, the less examination the firewall can perform. Generally speaking, lower-level firewalls are faster, but are easier to avoid.

### 3.1.1 Network Layer firewalls

Also called *Packet-filtering firewalls*, they validate packets based on protocol, source and destination IP addresses and port numbers as well as other parameters within the IP layer. This technology is commonly used in switches and routers and it is generally accomplished by using **Access Control Lists (ACL)**. Due to the low-level on which *network layer firewalls* work, they tend to be very fast and very transparent to users.

In a networking environment, **ACLs** allow or deny flow traffic through rules applied to port numbers or IP addresses that are available on a host or

other layer 3. They can generally be configured to control both inbound and outbound traffic.

The main problem of ACLs is that they are static, and packet-filtering firewalls do not have visibility into the data field of the IP packet. Another problem of network layer firewalls is that some attacks do not seem to be an attack for these firewalls, due to the attackers using well-known port numbers, or tunnels traffic unsuspectingly within the traffic allowed by the ACL rules. These caveats make Network Layer Firewalls a defense mechanism that is not fully trustworthy, but a complement to others.

### 3.1.2 Application Layer firewalls

Regarding the OSI model, the application layer is the highest layer it has. *Application Layer firewalls*, as their name suggests, work in the application layer of the OSI model. These devices act on behalf of a client for requested services, and provide additional information from port scans and application attacks.

But there are some caveats. The higher the OSI model goes up, the more software will be needed instead of hardware. This makes *Application Layer firewall*s slower than *Network Layer firewalls* because they have to run applications, maintain the state for both the client and the server, and inspect traffic at the same time. Another problem is that the firewall needs to know how to handle specific applications.

## 3.2 Firewalls against DoS and DDoS

Figure 3.1 shows the four most common targets of DoS and DDoS attacks and the percentage of attacks that achieve their goal. This graph is based on Radware Emergency Response Team (ERT)[27] security report[26] during 2011, 2012 and 2013.

After the internet pipeline and the servers, the third most common target of DoS and DDoS and also one of the most vulnerable ones (an average of 25% of attacks achieve their purpose) is the firewall. ACLs can store thousands of active connections to filter out legitimate and illegitimate traffic. The problem is that some DDoS attacks (HTTP floods, TCP SYN floods, etc.) are composed of millions of apparently legitimate items of traffic and the firewall opens a new connection on its ACLs for each malicious packet. This results in the quick exhaustion of the connection tables, and once they reach

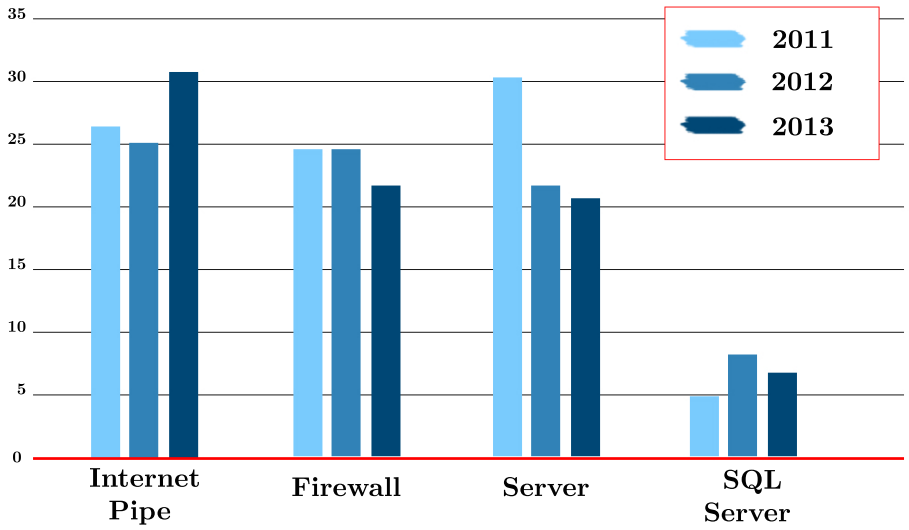



Figure 3.1: Network components being attacked

the maximum capacity, the firewall will drop all the new connections (from legitimate and illegitimate clients).

This happens due to the fact that firewalls are not designed to be a defense mechanism against DoS or DDoS. Firewalls match every packet one by one against the ACLs and since firewalls cannot distinguish between malicious and legitimate users, when an attack is carried out, it results in bottlenecks.

Although firewalls cannot distinguish between legitimate and illegitimate clients, there are some defense mechanisms based on firewall technologies and firewall complements against DoS attacks. In Section 4.1.2 we will go through some defense mechanisms against SYN flooding attacks, categorized according to whether defense mechanism is an **end-host** (see 4.1.2.1) or a **network-based** (see 4.1.2.2) mechanism.

## 3.3 SDN vs Firewalls

Previous section shows why firewalls are not the best solution against DoS and DDoS attacks. As it explains, since firewalls are focussed on filtering traffic based on flows rather than in packets, it becomes more difficult to discern discern between legitimate and illegitimate traffic. OpenFlow features

can play important roles to complement firewall technologies, listed here below:

- Section 2.2 explains the basic feature of SDN technology, the separation of the *control plane* and the *data plane*. This division between both levels allows the abstraction of the lowest-level network elements and technologies. As a result, it enables software developers with little knowledge in network programming development and design software using high-level languages to administrate network infrastructures. Moreover, vendors of classic routers and network components, integrate the control plane software in them, thus denying the possibility of changes. *Network-based firewalls* are integrated into normal routers and designed by vendors. This restricts their total customization according to customer desires.

  As explained in 2.1.2 DoS and DDoS may be carried out in different ways, depending on the purpose of the attack. *Protocol Attacks* (see 2.1.2.1), *Bandwidth attacks* (see 2.1.2.2) and *Protocol Attacks* (see 2.1.2.3) have their own properties and they are designed differently. Therefore, the same criteria should be applied to defense mechanisms. SDN allows for the design of specific defense mechanisms depending on the goals and vulnerabilities of the defending target.

- Both *Network-based firewalls* and SDN technology are based on network policies, which define the funcions to be carried out for the flow by means of table entries and with information to match against the traffic. Firewalls use ACLs and *OpenFlow Switches* use *Flow Tables* (see 2.2.1.1). The fields of the ACLs entries[2] are more limited than the *OpenFlow Flow Tables* ones. A typical table entry in firewall is *SRC IP, DST IP, PORT* and *ACTION*. On the other case, *OpenFlow Flow Tables* have more fields for different protocols and layers in the OSI model (MAC address, IP address, VPN, MPLS).

  It follows that with *OpenFlow Flow Tables* it is possible to define more concrete policies than with ACLs, enabling the defense of DoS and DDoS attacks targets by defining more specific characteristics in them (for example, by controlling the traffic sent to the 80 port of a webserver).

- SDN technology makes it possible orchestrate a set of *OpenFlow Switches* through one unique *Controller*, thus allowing for the centralization of the network's control plane. Unlike *Network-based firewalls*, which are set up for each switch or router in the network topology, *OpenFlow*

*Protocol* provides the advantage of administering all the traffic and the flow policies through only one server, thus allowing the scalability and reliability over the network infrastructure.

Thanks to this orchestration, it is easier to know about DoS and DDoS since the *Controller* can trace all the traffic throughout the *OpenFlow Switches* and follow it as a whole.

# Chapter 4

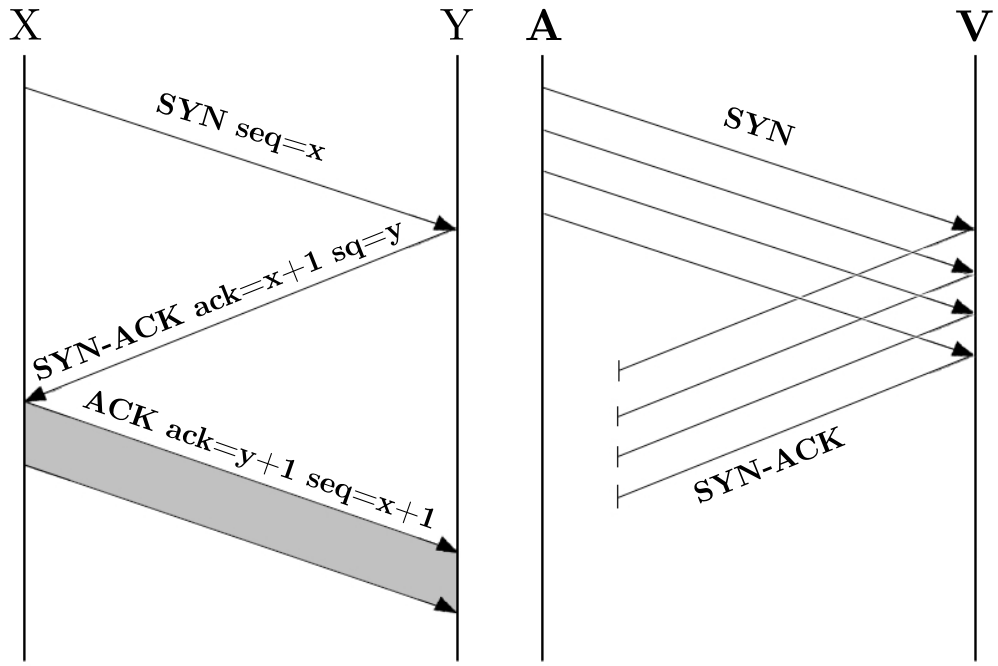
# Theoretical Analysis

## 4.1 Protocol Attacks: TCP SYN Flooding

TCP (Transmission Control Protocol) is one of the most common protocols within the transport layer, and one of the central components of the Internet Protocol suite (IP). TCP provides reliable, ordered, error checked streams of packets between two hosts. In addition to these characteristics, TCP is a connection-oriented protocol, that is, a prior connection between both parties is necessary before starting the exchange of information. This process is known as **TCP three-way handshake** (Figure 4.1(a)).

*X* is a client that wants to carry out a friendly TCP connection with the server *Y*. First of all, *X* makes a request by sending a synchronize *(SYN)* message to *Y*. The *sequence number* field of the TCP header gets a value, *x*. The server receives the request and responses by sending an acknowledgment *(SYN-ACK)* back to the client, in which the *acknowledgement number* is the *sequence number* of the synchronize message increased by one (*x+1*); and the *sequence number* is a new value *y*. Once the client receives the *SYN-ACK* it checks if the acknowledgement number received matches with the sequence number sent with the *SYN* message incremented by one. When the client carries out the necessary verifications it sends back an *ACK* message, with the *sequence number* field sets as *SYN-ACKs acknowledgement number* and the *ACK* message *acknowledgement number* field is set to *y+1*.

While the server waits for the *SYN-ACK's* response, it keeps the connection in a half open state and maintains a backlog queue for the information about the connections. Once the server receives the *ACK*, it changes the state to *established* and frees up memory of the queue. Because the size of the backlog queue is finite, the half-open connection will remain on it until a time-out is exceeded. In the case that the queue is full, all new incoming




(a) TCP Tree-way handshake (b) SYN Flooding

Figure 4.1: TCP Three-way handshake and SYN Flooding

connection requests will be dropped.

Once the TCP three-way handshake process is finished the connection between client and server is established, and they already can share information between them. To close the connection between both parties, there are two ways: once the the client has finished it requesting to the server, it sends a *FIN* message and the server will answers with a *FIN ACK* message; the other way is when some error occur, then either the server or the client send a *RST* message to conclude the connection.

**TCP SYN Flooding attack**, as the name suggests, aims to exhaust the server's backlog queue flooding it with *SYN* messages, but once the attacker receive the corresponding *SYN-ACK* from the server, they will not response with the *acknowledge* message, forcing the server to keep the connection information in the backlog queue until the time-out is exceeded (Figure 4.1(b)). As a result, when a *friendly* client wants to set up a TCP connection with the server, it will be denied. The biggest challenge for the attacker is ensure that the source IP address which is used to establish the connection is not reachable by the server. Otherwise, the source will send a *RST* packet to the

server that cause server to reset the connection and free up the memory of the queue. The attackers try to achieve this goal through different methods of attacks, explained in the section below (see 4.1.1).

### 4.1.1 Methods of Attack

Can be categorized according to how the attacker carries out the attack over the victim: Direct Attack, Spoofed-based Attack and Distributed Attack [12].

#### 4.1.1.1 Direct Attack (Figure 4.2)

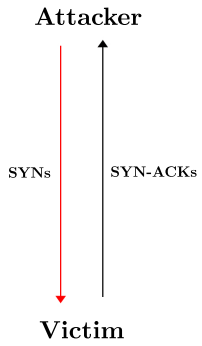



Figure 4.2: Direct Attack

In this case, the attacker is the one that accomplishes the attack against the victim directly. It does not even spoof the source IP address. Instead, it will just ensures that there will not be response after it receives the *SYN-ACK* message.

#### 4.1.1.2 Spoofed-based Attack (Figure 4.3)

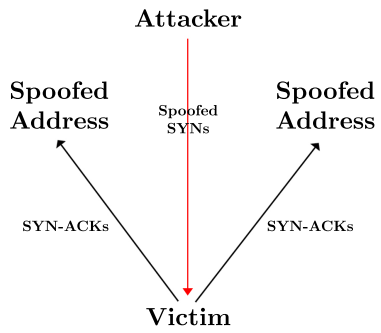



Figure 4.3: Spoofed-based Attack

This version of SYN flooding attacks the victim directly, but it spoofs the source IP address in the *SYN* packet. As has been explained before, a primary consideration is address selection. An attacker can choose spoofed IP address either using a single source which is known will not respond to the *SYN-ACK*, or using a list of source address under the assumption that some percentage of them will not respond.

#### 4.1.1.3 Distributed Attack (Figure 4.4)

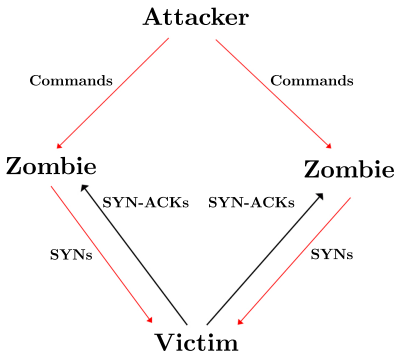



Figure 4.4: Distributed Attack

In this case, the attacker carries out the attack through numerous zombie machines on the internet. This attack is much more difficult to counter, due

to the attacker not being the one that accomplish. Zombie machines are constantly added and removed from the botnet.

### 4.1.2 Prevention and Response

The **TCP SYN Flooding** vulnerability has been well-known for over a decade. This section discusses several solutions aimed at making these attacks ineffective. Some of these are already on the market, but no solution has been standardized yet.

These solutions are divided in two classes depending on where the defense mechanisms are being implemented. The first class involves modification and improvement of the end-host TCP implementation, it is called **End-host Countermeasures** (see 4.1.2.1). The second class involves implementation in the network itself, **Network-based Countermeasures** (see 4.1.2.2).

#### 4.1.2.1 End-host Countermeasures

**Increasing TCP Backlog** The basic TCP SYN Flooding attack aims at overflowing the host's backlog of connecting sockets in order to deny future connections. A basic countermeasure is **to increase the size of the TCP backlog for incoming connections**. Increasing the size of the backlog is typically possible by altering the *listen()* method and setting operating system kernel parameters.

This method should not be considered as a good defense mechanism against SYN flooding attacks, because an attacker may scale to a larger number of orders than the host my support, even if the size of the backlog has been increased.

**Reducing the SYN-RECEIVED Timer** When the server receives the *SYN* message from a host, it keeps this connection in a *half-open connection* state until either the *ACK* is received or a timeout is exceeded; then the backlog memory is freed up. A simple mechanism is to **increase the number of resources devoted to *half-open connections* and to put a tighter limit on the timeouts**.

As in the method above, the flaw in this one is also when aggressive attacks occur, because the attacker can avoid the defense mechanism simply by increasing the rate of the arriving packets and, as a result, exhaust the backlog.

**SYN Caches** This method functions by reducing the amount of state initially allocated to backlog generated by a received *SYN* message, while putting off instantiating of the full state [15]. **A hash table with a limited amount of space in each entry is used to store the connection information state instead of using the backlog**. The hash table keeps the information until all the connection process is finished and the server has received the handshake-completing *ACK*. Then, this data can be moved into a full backlog.

The code must be prepared to handle state overflows and choose which items to drop in order to preserve fairness. By taking care of these possible problems, the SYN cache data structure will be robust against attackers attempting to overflow its buckets, because it uses the initiator's local port number and some secret bits in the hash value.

**SYN Cookies** As with *SYN Caches*, *SYN cookies* also work by reducing the amount of state initially allocated to the backlog. But unlike cache approach, this technique causes zero state to be generated by a received *SYN* message. Instead of keeping the information in the server machine, this method **compresses the connection state into the bits of the *Sequence Number* TCP header field used in the *SYN-ACK* message**. When the server receives the *ACK* message with the sequence number increased by one, the data for the backlog can be generated and a full backlog can be instantiated by decompressing the *Acknowledgement* field. The decompression can be effective even under an SYN flooding attack because there is no storage load on the listener [7].

Not all the backlog data can fit into the 32-bits *sequence number* TCP header field, so some TCP options required for high performance might be disabled.

Some defense mechanisms might combine two or more of these defense mechanisms. For example, an option which is already included in some operating systems is the use of combined *SYN Cookies* and *enlarged backlog*. In a normal situation, SYN Cookie mechanisms are not enabled, but they are usually turned on when the backlog is running out of space.

#### 4.1.2.2 Network-based Countermeasures

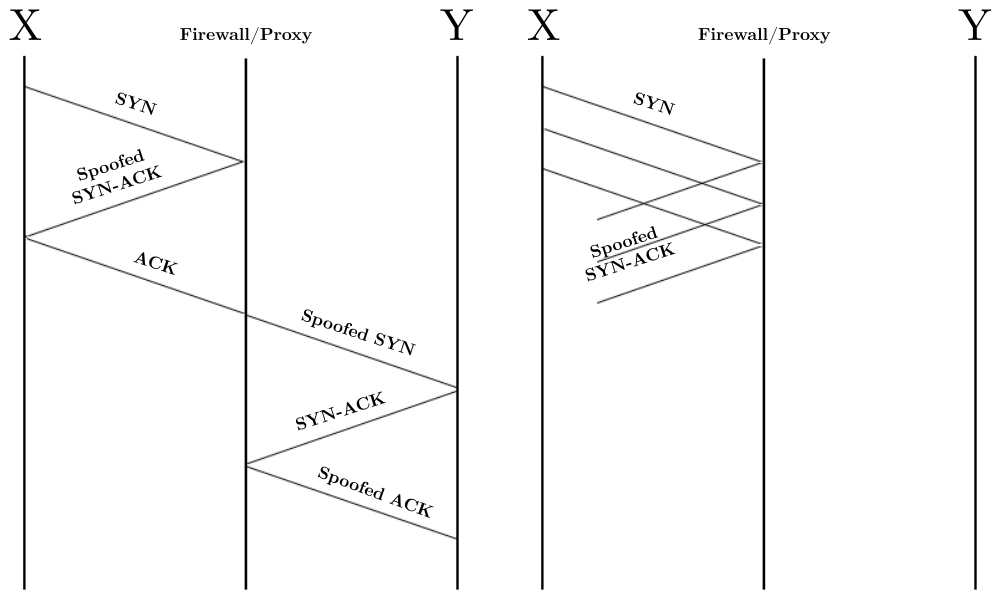
**Filtering** It works by using ingress and egress filtering **configuring external and internal interfaces on routers to block packets that**

**have source addresses from the internal network or outside the internal network**. This mechanisms is highly effective to prevent attacks that rely on spoofed packets.

However, it is not effective against attacks that using a botnet, in which each bot carries out the attack through a *Direct Attack* (see 4.1.1.1).

**Firewall and Proxies** These devices can buffer end hosts from SYN flooding attacks by means of two different methods, either using the **Firewall as a Relay** or using the **Firewall as a Semi-transparent Gateway** [29].

In the **Firewall as a Relay** method (figure 4.5), when the firewall receives the *ACK* message from client *X* before the packet reaches server *Y*, it responds to *X* with a spoofed *SYN-ACK* message. In a **fairness connection** (figure 4.5(a)), client *X* will respond with an *ACK* message and once the *firewall* receives it, it will send to server *Y* a spoofed *SYN* message and while forwarding a *ACK* message for establishing the legitimate connection.



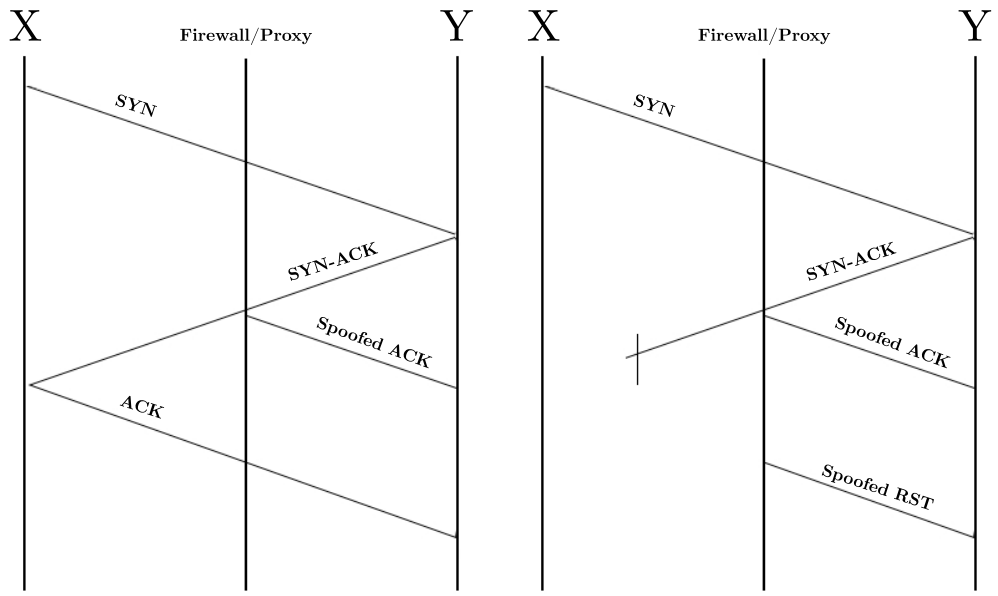

(a) Non-attack behaviour (b) Attack behaviour

Figure 4.5: Firewall as a Relay

If the server is **under attack** (figure 4.5(b)), since the *server* does not receive any *SYN* message until the *firewall* does not receive the *ACK* message, no half-open connection are kept in the backlog of server *Y*.

Figure 4.6 shows another case, where the **Firewall acts as a Semi-transparent Gateway**. In this case, once the *firewall* has received *SYN* message, it is forwarded to the server *Y*. This server sends the *SYN-ACK* sends back to *X* message and, as the *firewall* receives it, two functions are carried out: the firewall forwards the *SYN-ACK* message to *X* and it also spoofs an *ACK* message and sends it to *Y*. In a **secure connection** (figure 4.6(a)), client *X* will respond with the *ACK*.

If *X* is making an **attack** (figure 4.6(b)), it will not respond with the *SYN-ACK*, and after a timeout, the *firewall* will respond with an spoofed *RST* message. This method intends not to keep half-open connections in the backlog, but to rather keep the state as an established connection.




(a) Non-attack behaviour (b) Attack behaviour

Figure 4.6: Firewall as a Semi-transparent Gateway

**Active Monitor** This category of solutions consists of using a software agent to continuously monitor TCP/IP traffic in a network at a given

place. An agent can collect communication control information to generate a view of all connections that can be observed on a monitored network.

# Chapter 5

# Experimental Implementation

This chapter presents a DoS defense mechanism against Protocol Attacks, specifically **SYN Flooding Attacks**.

As mentioned int the Introduction (Chapter 1) the scope of this project is to study how *SDN* technology can help to defeat DoS and DDoS. It will also be compare SDN with firewall technologies, and analyse how they behave under DoS and DDoS attacks.

This study presents a *Network-based defense mechanism* developed with the *POX Controller*. In order to test this mechanism, a virtual scenario is simulated, using the following technologies:

- It is necessary to work over on virtual network where tests may be carried out. To do so, *Mininet* is used (see 5.2.1) to simulate a network topology, as explained in subsection 5.3.1.
- Through a *POX Controller* an algorithm is developed, as shown in subsection 5.1;this algorithm works as the defense mechanism against *TCP SYN Flooding*.
- In order to get information about the traffic flow in the network and to obtain statistics on the amount of packets flowing to each host, *sFlow* (see subsection 5.2.2) is used to monitor and control the traffic on the network.
- Some virtual host of the network topology will work as *attackers*. Their behaviour is explained in subsection 5.3.2

Next chapter (see 6) shows the results obtained once the tests have been carried out.

## 5.1 Algorithm proposed against TCP SYN Flooding Attacks

Algorithm 1 shows the defense mechanism behaviour this thesis is focussed on. It is based on the idea that if a client has ever carried out a correct connection with the server, future connections from this same client will also be legitimate ones. As it is explained in subsection 2.1.3, there are four different types of defense mechanisms, depending on the phase the attack is on. The defense mechanism shown by the algorithm uses two different types of defense mechanisms: *Attack Detection* (see 2.1.3.2) and *Attack Reaction* (see 2.1.3.4).

For the *attack detection phase*, the defense uses anomaly detection to warn whenever an attack is taking place. The anomaly detection system of this defense mechanism tries to identify the attack based on two possible anomalies. The first one is checking out the TCP header flags; when an attacker tries to perform a SYN Flooding Attack by spoofing the packets, it is common to modify the packet fields, both in the IP (see A.1) and TCP headers (see A.2). Sometimes TCP flags are modified to confuse the target [10, 16]. In this case, the algorithm will only take into account the packets with a correct form of SYN, SYN ACK and ACK messages, skipping the ones with fake fields. The second possible anomaly depends on the amount of traffic arriving at the server. If the number of packets for the server is over the threshold, then the defense mechanism is informed about it and appropriate measures are taken.

Once the server is under attack, the *attack reaction phase* starts, by denying malicious traffic from untrusted sources. Only the ones operating in a correct way will be able to communicate with the server.

As subsection 2.2.3 explains, when the packets get to the switch and do not match the flow entries, they are sent to the *Controller*. In this case, the *packet_in* variable defined in the algorithm represents those packets. The *Controller* uses different data structures and variables to control the network. In order to control the no attacker hosts, three different data structures are used: *SYN_Received*, *SYNAKC_Received* and *Friendly_Hosts*. Through the variable *Flag_Attack* the *Controller* is able to know if the system is either under attack or in a normal state. The following lines explain the different behaviours of the mechanisms in both cases.

*A* The target of the algorithm in a **non-attack state** is to keep the information of the legitimate hosts in order to recognize them in future

**input** : *packet_in* arrived from the OpenFlow Switched
**output**: *action* to carry out
**Data**: Dictionaries: $SYN\_Received$, $SYNACK\_Received$, $Friendl\_Hosts$, $Flag_Attack$

```
1  if no Flag_Attack and packet_in protocol is TCP then
2      if packet_in is SYN message then
3          Add to SYN_Received dictionary, with index = IP source
           and value = seq and ack
4          ;
5      else if packet_in is SYN ACK message then
6          if packet_in destination IP is in SYN_Received and
           ack + 1 = seq SYNACK_Received then
7              Add to SYNACK_Received dictionary, with index = IP
               destination and value = seq and ack
8              ;
9          ;
10     else if packet_in is ACK message then
11         if packet_in source IP is in SYNACK_Received and
           ack + 1 = seq SYN_Received then
12             Add to Friendly_Hosts dictionary, with index = IP
               destination and value = port
13             Install flow entry in OpenFlow Switch in both directions
               ;
14         ;
15 end
16 else if Flag_Attack and packet_in protocol is TCP then
17     if packet_in IP is in Friendly_hosts then
18         Forward packet_in
19         ;
20     else  Drop packet_in ;
21 end
22 else
23         Forward packet_in
24 end
```

**Algorithm 1:** Algorithm against TCP SYN Flooding Attacks

connections. These hosts will be stored in the *Friendly_Hosts* list, keeping their IP and the *OpenFlow Switch* through which they arrived. In order to accomplish this classification, *packet_in* fields are analyzed and the TCP Handshake process is performed as follows.

Once the algorithm has checked that the *Flag_Attack* variable is *false* and that the *packet_in* transport protocol is TCP, it ascertains, through the *TCP Flags field*, the type of message. Let's think on a secure TCP connection: the first message must be a *SYN* message, and that is proved through the line *2* of the algorithm. Then, the source IP and the values of the *acknowledgement* and *sequential number* fields (line *3*) are stored in the *SYN_Received*. After that, the packet will be forwarded to its destination. Once the server receives the *SYN* message, it will respond as it is explained in section 4.1. The *Controller* will receive the *SYN ACK* message from the server in line *6*. In this part, the main idea is to move the IP source address, stored the *SYN_Received* message list, to the *SYNACK_Received* message list. In order to ensure a secure connection, it takes the corresponding values in the *SYN_Received* list to the destination IP address of the *packet_in*, and it checks the values of the *sequential* number and the new *acknowledgement* value (see line 6). The next step is the most important one since to it will be checking if the client is able to respond with an *ACK* message. Just like before, the *Controller* will check if the *ACK* has received matching values in the *SYNACK_Received* list, in order to generate a legitimate IP address list, called *Friendly_Hosts*.

After it has been checked that it is a *friendly* client, two *flow entries* will be set up in the *flow table entries*, in both directions. The matching fields of these flow entries are:

- The transport protocol must be TCP.
- The port of the server must be 80 (web server).
- An idle timeout is set up to 60 seconds in order to free up memory at the flow tables.
- In one flow entry, the source IP address must be the client IP address and the IP destination address, that of the server. In the other flow entry, it should be vice versa.
- The action to be accomplished in each flow entry is to forward the packets through the *OpenFLow Switch* port of the destination IP address.

Once the *flow entries* are installed in the *OpenFlow Switch*, all the traffic that matching the *flow entries* will not be sent to the Controller, and the *OpenFlow Switch* will carry out the appropriate actions. If no flow traffic matches with the *flow entries* in less than 60 seconds, they will be deleted. If that happens, every time a source IP address contained in the *Friendly_Hosts* does not follow the steps indicated here below, both *flow entries* will be directly set up again.

Moreover, *sFlow* is continuously controlling the amount of traffic flowing to the server. A threshold is defined in order to warn the *Controller* that an attack is being carried out. To do so, *sFlow* will send a message through a *JSON API* informing about the attack. Once the attack has been detected, *Flag_Attack* variable changes to *true* and the algorithm will behave as follows:

*B* The limit has been exceeded and the *Flag_Attack* has changed to *true*. This means that the server is **under attack**. Once the server knows that is under attack and has figured out which host has not carried out the attack, the *Attack Reaction* mechanism is quite simple. The condition state on line *16* checks whether the *Flag_Attack* is true and the *packet_in* protocol is TCP. The following lines show two very simple conditions. If the *packet_in* source IP is contained in the *Friendly_Hosts*, both flow entries will be installed as it would be done in a normal situation. But if is not contained in it, all the *packet_in* with IP server destination will be dropped.

In the same way as *sFLow* warns the *Controller* about overcoming the threshold, it will also inform it when flow traffic is again in a normal state, under the limit. In this case, the *Flag_Attack* variable changes to false and the algorithm behaves as explained above.

## 5.2 Tools

### 5.2.1 Mininet

**Mininet**[17] is able to create, on a single native or virtual machine, realistic virtual networks running in real kernels, switches and applications code. **Mininet** allows developing and experimenting with *OpenFlow* and *Software Defined Networking* systems. The creation and maintenance of the network topology used in this experiment (see subsection 5.3.1) has been done through Mininet.

### 5.2.2 sFlow

sFLow[31] is used to monitor and manage traffic in complex networks. It can be used in virtual networks but also in real ones. It has a *JSON API*[30], which has been used in this project to inform the *Controller* of the network situation.

## 5.3 Environment

Once the defense mechanism has been explained in detail, it is important to carry out the necessary tests in order to study its behaviour. Consequently, a virtual network topology has been implemented through **Mininet** with the purpose of simulating a real environment. Such topology is explained in detail in section 5.3.1.

In that topology some of the hosts will behave as attackers. Their mechanism is explained in section 5.3.2.

### 5.3.1 Topology

Any system connected to the internet and providing TCP-based network services, such as a Web server, FTP server, or mail server, is a real target for **TCP SYN Flooding attacks**[32]. In this implementation, a *Web Server* has been used as the victim of the attack.

Figure 5.1 shows a simple topology created through **Mininet** in a virtual environment. This topology is initialised by executing the following command line,

$ *sudo mn –topo SINGLE,7 –mac –switch ovsk –controller remote*

which means:

- *–topo SINGLE,7*: Simple topology with seven *hosts* directly connected to the *OpenFlow Switch*. One of them behaves as a web server responding to the host's GET requests (once the TCP connection is correctly done) with a simple *.html* document.
- *–switch ovsk*: One *OpenFlow Switch*.
- *–controller remote*: A remote *POX Controller* connected to the *OpenFlow Switch*.

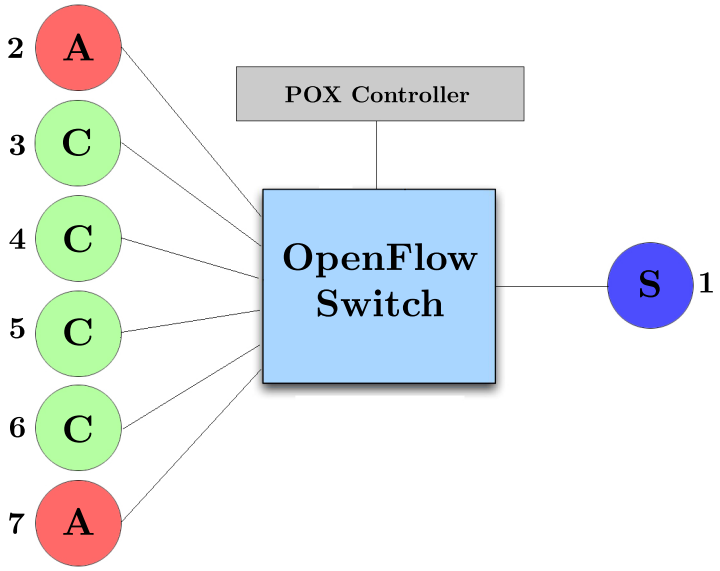



Figure 5.1: Mininet topology

The red circles in the figure 5.1 represent the attacker, whose attack mechanism is explained in the next section. The green ones are legitimate clients requesting data to the server (blue circle) after the TCP connection has been established.

### 5.3.2 Attack Behaviour

As section 4.1.1 explains, to carry out a **SYN Flooding** attack the attackers have made sure that once they receive the *SYN ACK* message, they will not respond with neither a *ACK* message nor a *RST* message, this forcing the server to keep their information in a half-open connection state.

The selected attack to achieve this behaviour is a **direct attack** (see 4.1.1.1). Through a *Python* implementation, the developed attack software aims at creating a different socket each time it is sent to the *server*.

It has created all the field values for both *IP* header and *TCP* header which are then merged into one packet. In order to flood the server, the attacker will send 200 *SYN* messages per second.

# Chapter 6

# Results

Regarding the main aspects of DoS attacks and their defense mechanism, particularly **TCP SYN Flooding attacks**, they could be summarized as follows: DoS and DDoS aim at crashing the system either by sending crafted packets that exploit a software vulnerability or by ***sending massive volumes*** of useless traffic to overwhelm the target. Protocols have some vulnerabilities that are well known by the attackers, who try to exploit the system by taking into account the expected behaviour of the protocols. In an SYN Flooding situation, the attacker aims at making the server keep a half-open connection in the ***server backlog***, in order to deny service to legitimate users.

This project presents a defense mechanism where legitimate users who have carried out a secure connection, will be able to connect to the server, even if there are attacks in process. To do so, the *OpenFlow Controller* will generate a ***legitimate users list*** where the entries are data from each host that accomplishes the TCP connection with the server.

Throughout this chapter an attack scenery will be shown as well as how the network environment reacts to it. In order to study how TCP SYN Flooding attacks react over the system and how the defense mechanisms responds, the following information will be shown and discussed in this chapter.

Two different scenarios will be simulated. On the first one, there is no defense mechanisms; instead, the *Controller* will forward all the packets it receives to their destination (see section 6.1). On the second case, the defense mechanism is set up at the *Controller* (see section 6.2). For both scenarios and regarding figure 5.1, the behaviour of the hosts will be:

- *Hosts 3, 4, 5, 6* are the legitimate users. Each one of them will continuously do a GET request to the server in different intervals of time. Hosts *5* and *6* will request data each 125 and 130 seconds respectively,

in order to show that even if the *flow entry* has been deleted from the flow table (due to the *idle time*), the legitimate hosts will still be able to accomplish the connection.

- *Hosts 2 and 7* are the attackers. They will start the attack 135 seconds after the test has started.
- *Hosts 1* is the web server.

The next sections will show the amount of packets per second reaching the *server* in both cases (with and without defense mechanism), as well as the state of the server backlog, the data of the Friendly_Hosts list and the screenshots of a legitimate host terminal carrying out GET request to the server (appendix B).

## 6.1 Attack without a defense mechanism

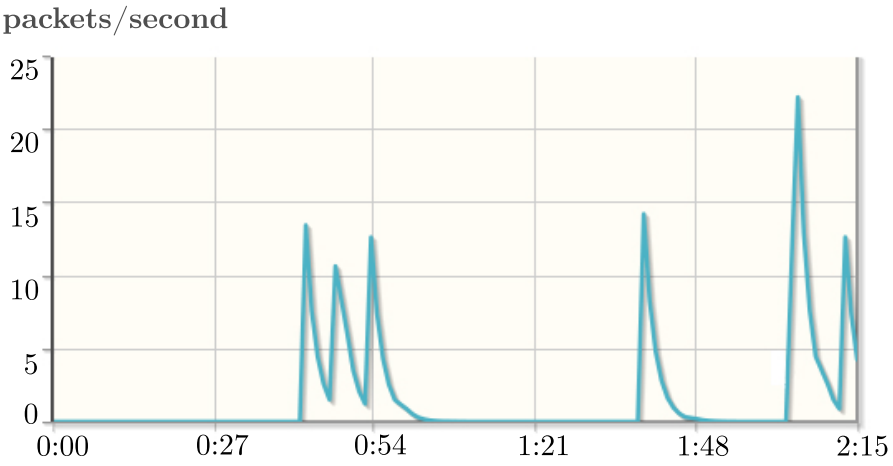
Figure 6.1 shows both the traffic generated to the server by legitimate hosts (figure 6.2(a)) and the state of the connections in the server backlog (figure 6.2(b)) before the attack.

Both figures show a normal behaviour of a web server. The server receives no more than 25 packets per second and the state of the connections is *TIME_WAIT*, which means that the server is connected to the hosts and it is waiting for data requests. The backlog shows that hosts *3*, *4*, *5* and *6* have achieved the connection.

135 seconds hosts later, *2* and *7* will start the attack. Each one of them will send an average of 200 packets per second to the server, all of them *SYN* messages. Once both hosts have started the attack, the server will receive more than 500 *SYN* messages, changing their ports in each message. Such situation is shown in figure 6.2(a).

The server has not any TCP SYN defense mechanism so it will crash at some point as expected. As the figure shows 6.2(a), 470 seconds after the attack has started, the server will stop receiving data from the hosts. This means that the attackers have reached their scope, which is to fill the backlog memory up in order to deny service to legitimate users. Figure 6.2(b) shows the connection states at the backlog where a large number of state connections from hosts *2* and *7* in a *SYN_RECV* state will show. *SYN_RECV* states means that the server has responded to the *SYN* message received with a *SYN ACK* message and it is waiting for a response.

In order to show the legitimate host behaviour before and after the attack has been launched, figures B.1(a) and B.1(b) are attached in appendix **??**. Figure B.1(a) presents the normal behaviour of a host requesting data, but in figure B.1(b) it can be seen that the legitimate host is not able to reach the server.




(a) Flow traffic arriving to the server

```
root@mininet-vm:~# netstat -n -p TCP tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID
tcp        0      0 10.0.0.1:80             10.0.0.3:49340          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.4:53162          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:49345          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.6:41191          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.5:44659          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:49344          TIME_WAIT   -
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name    Path
```

(b) Server backlog

Figure 6.1: Normal traffic before the attack without a defense mechanism


(a) Flow traffic reaching the server

```
root@mininet-vm:~# netstat -n -p TCP tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID
tcp        0      0 10.0.0.1:80             10.0.0.7:20655          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:29313          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:59819          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:8269           SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:26301          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:16796          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:15755          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:8270           SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:17076          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:20566          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:28520          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:10692          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:40859          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:31784          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:43978          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:56706          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.3:49411          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.5:44719          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.6:41258          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:49413          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.4:53222          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:49400          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:49404          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.4:53230          TIME_WAIT   -
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name    Path
root@mininet-vm:~#
```

(b) Server backlog

Figure 6.2: Attack scenary without a defense mechanism

## 6.2 Attack with defense mechanism

Once it has been shown that the server has crashed under the attack in absence of a defense mechanism, throughout this section we will study how the *Controller* responds to the attacks.

Figure 6.3 shows the same situation as in the previous case: the way the environment behaves in a normal situation where legitimate hosts request data to the server. As the section above explains, the maximum amount of packets per second reaching the server is almost 25 (figure 6.3(a)) and the backlog connections state situation (figure 6.3(b)) is normal, without half-open connection states.

The *Controller*, when there is not any attack, functions by keeping the information of the legitimate hosts following the *TCP three-way handshake* process. This process works with three lists as it is explained in 5.1 Here below it has followed this process with host *3*, showing the content of the lists at each phase of the connection:

**1. Host *3* sends a SYN message to host *1* (server)**. Host *3* sends a *SYN* message to the server in order to carry out a TCP connection. In the TCP fields it includes the sequential (*seq = 1789773694*) which is stored in the *SYN_Received* list with its IP address.

```
-------------------------------------------------------
TCP PACKET SYN  10.0.0.3 --> 10.0.0.1 :
        seq = 1789773694 ; ack = 0

        SYN RECEIVED {'10.0.0.3':
                {'ack': 0, 'seq': 1789773694}}
        SYN ACK RECEIVED {}
        FRIENDLY HOSTS {}
-------------------------------------------------------
```

**2. Server responds with a SYN ACK message to host *3***. The server responds with a *SYN ACK* message the *SYN* message sent by the host. In order to move the content of the *SYN_Received* list to the *SYNACK_Received* list, the algorithm checks if the *ack* value (*ack = 1789773695*) is the sequential number of the *SYN* message plus one (*seq = 1789773694 + 1*). In this list, it is include the IP address and the new *ack* number as well as the new *sequential* value of the message. The state of the *SYNACK_Received* list is shown below:

```
------------------------------------------------------------
TCP PACKET SYN and ACK 10.0.0.1 --> 10.0.0.3 :
        seq = 3772125158 ; ack = 1789773695

        SYN RECEIVED {}
        SYN ACK RECEIVED {'10.0.0.3':
                {'ack': 1789773695, 'seq': 3772125158}}
        FRIENDLY HOSTS {}
------------------------------------------------------------
```

**3. Host *3* responds with an ACK message to the server**. In the third and last step of the three-way handshake TCP connection, once the *Controller* receives the *ACK* message from the client (as a *ACK SYN* response), it checks both *sequential* number and *ack* number to compare it with the ones in the *SYNACK_Received list*. In order to carry out the connection, *ack* value of the *ACK* message (*ack = 3772125159*) must be the *sequential* number of the *SYNACK_Received list* plus one (seq = 3772125158). If these fields do not match, the server will send a *RST* message to the client.

```
------------------------------------------------------------
TCP PACKET ACK : CONNECTION SUCCESFULL 10.0.0.3
        seq = 1789773695 ; ack = 3772125159

        SYN RECEIVED {}
        SYN ACK RECEIVED {}
        FRIENDLY HOSTS {'10.0.0.3': 3}
------------------------------------------------------------
```

Host *3* has managed to connect to the server and now is in the *Friendly_Hosts* list with the *OpenFlow Switch* port as the IP value. Seconds before, the attacks started and once all the legitimate users have connected with the server, the content of the *Friendly_Hosts* list would be:

```
------------------------------------------------------------
        SYN RECEIVED {}
        SYN ACK RECEIVED {}
        FRIENDLY HOSTS {'10.0.0.5': 5,
                        '10.0.0.4': 4,
                        '10.0.0.6': 6,
                        '10.0.0.3': 3}
------------------------------------------------------------
```

It is also important to remark that for all the host that have demonstrated to be legitimate hosts, the flow entries have been installed in the *OpenFlow Switch* in order not to send more packets to the *Controller* from those hosts. Section B shows the *flow tables* installed in the *OpenFlow Switch*.

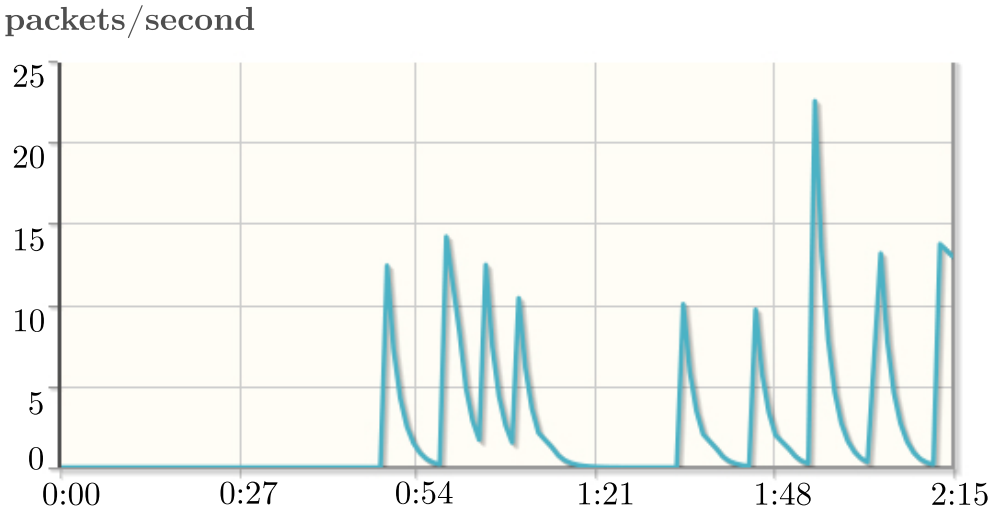
As in the example in section 6.1, hosts *2* and *7* start the attack 135 seconds after the test has begun. Figure 6.4 shows both the traffic flowing to the server and the server's backlog. When the rate of packets to the server increases beyond the set threshold, *sFlow* sends an event to the *Controller* which starts to drop packets from illegitimate hosts. This process takes less than 5 seconds as it is shown in figure 6.4(a).

After the defense mechanism has been activated, legitimate hosts can still send and receive information to and from the server. This is because of the *Friendly_Host* list. The state of the lists would be:

```
------------------------------------------------------------
        SYN RECEIVED {}
        SYN ACK RECEIVED {'10.0.0.7':
                                {'ack':1, 'seq':2057411521},
                          '10.0.0.2':
                                {'ack':1, 'seq':1159457829}}
        FRIENDLY HOSTS {'10.0.0.5': 5,
                        '10.0.0.4': 4,
                        '10.0.0.6': 6,
                        '10.0.0.3': 3}
------------------------------------------------------------
```

While hosts *3*, *4*, *5* and *6* are in the *Friendly_Host* list, hosts *2* and *7* are in the *SYNACK_Received* list because the *Controller* has not received any *ACK* message after the server sent the *SYN ACK* message. Because of that, these hosts are considered to be illegitimate and all the packets sent from them will be dropped to prevent the server's backlog from filling up.

Figure 6.4(b) shows the server's backlog few seconds after the attack has been blocked. There is not any anomaly, only a small set of packets in a *SYN_RECV* state, which do not interfere with the normal process of the server.

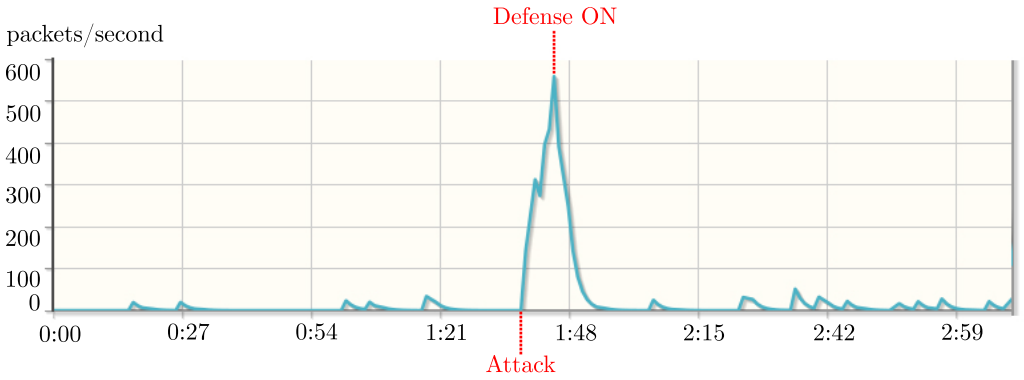



(a) Flow traffic reaching the server

```
root@mininet-vm:~# netstat -n -p TCP tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID
tcp        0      0 10.0.0.1:80             10.0.0.3:42469          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.6:42472          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42484          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42477          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42486          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.4:59100          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42474          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.6:42482          TIME_WAIT   -
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name    Path
root@mininet-vm:~#
```

(b) Server backlog

Figure 6.3: Normal traffic before the attack with a defense mechanism




(a) Flow traffic reaching the server

```
root@mininet-vm:~# netstat -n -p TCP tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID
tcp        0      0 10.0.0.1:80             10.0.0.7:20655          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:29313          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.7:59819          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:8269           SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.2:26301          SYN_RECV    -
tcp        0      0 10.0.0.1:80             10.0.0.3:42884          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.6:43244          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42894          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42916          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42900          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42910          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42897          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42913          TIME_WAIT   -
tcp        0      0 10.0.0.1:80             10.0.0.3:42906          TIME_WAIT   -
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name    Path
root@mininet-vm:~#
```

(b) Server backlog

Figure 6.4: Normal traffic before the attack without a defense mechanism

# Chapter 7

# Conclusions

In this chapter we will discuss opportunities for improvement in Section 7.1 and we will review the work done for the final project in Section 7.2.

## 7.1 Future work

In order to study how SDN technologies can help by defeating DoS and DDoS attacks, throughout this project a defense mechanism for protocol attacks has been presented and tested. Nonetheless, due to the complexity and extension of the matter, some non tested improvements arise here:

- As section 2.1.2 shows, we have classified three different types of DoS attacks depending on the target of the attackers. In this project only one kind of protocol attack has been studied, in particular **TCP SYN Flooding attacks**. By means of getting general conclusions about how the OpenFlow standard can help by defeating these kind of attacks, a particular survey for each DoS attack would be interesting.

- The software developed in the POX Controller environment aims at offsetting the TCP handshake connection process vulnerabilities. Nonetheless, this application is meant specifically for this type of attack. Some other protocol attacks exist and particular software for each of them is needed in order to defeat them.

- There are four different phases when a DoS or DDoS attack is to be defeated. These phases are explained in section **??**. A defense mechanism might include all of them or only one, but the more of these phases a defense mechanism has, the higher the level of security and complexity it will have. In the proposed mechanism, there are two of these phases: *attack detection* (based on anomaly detection) and *attack reaction*.

In order to fully analize SDN features against DoS attacks, a defense mechanisms including all the phases would be of interested.

- Even though *Mininet* creates virtual network topologies by helping to test OpenFLow software, it has some limitations. Assigning desired resources to the hosts, spoofing IP addresses and particular flow analysis are some of them.

  In order to carry out a comprehensive study, it would be interested to use a real network topology with physical devices. For example, by using specific PCs as the *Controller* and the web server.

## 7.2 Discussion

In this project we present a DoS defense mechanism working in SDN technology, by means of an OpenFlow Protocol. In order to study and test how our work can help to prevent DoS and DDoS attacks, we have tested it in a virtual environment with Mininet, and we have obtained data results and graphics with the aid of *sFlow*.

Throughout this section, we will discuss the results obtained once the test has been carried out. There is also a comparison between the advantages and disadvantages of using either SDN technology or firewalls as DoS and DDoS defense mechanisms.

With the purpose of studying the behaviour of SYN Floodig attacks on devices that use TCP as their transport protocol and how to prevent them from these attacks, we have done two different tests and the results are presented in chapter 6.

*The first one* simulates a normal situation where there is no defense mechanisms, only a typical router which forwards the traffic to its destination. After some time with trustful requests from different hosts, the attack mechanism is launched from two anywhere in the topology hosts. Since there is no defense mechanism set up anywhere in the topology (SYN Cookies, SYN Cache, Firewalls, Proxies, etc.), the results has been as expected. Once the server is under attack, the state of the server backlog shows how it is filling up with *SYN_RECV* states; this means that the server is keeping the attacker request connections in half-open connection states. These connections remain in the backlog until a timeout is exceeded, then the server proceeds to discard them. If the rate of received *SYN_ACK* messages is too high, the backlog will fill up with half-open connection states. As chapter **??** explains,

when the backlog is full, the server starts to discard new request connections, even the ones coming from legitimate hosts.

The aim of this project is to show how SDN technology can help us to prevent this attacks. *The second test* presented uses the same topology and attack mechanism that the previous one, but in this case a defense mechanism is set up in a OpenFlow Switch. This defense mechanism has two phases, an *attack detection* phase, based on anomaly detection, and a *attack reaction* phase.

The *attack detection* phase uses two mechanisms in order to detect anomalies in the environment. One of them checks the TCP flags field to avoid malformed packets. The other one works with *sFlow* to inform the *Controller* about high amounts of traffic arriving to the server. Both mechanisms work correctly as explained in section 6.2, which shows how the server backlog is not full of half-open connections and how the packets from the attackers are dropped in the *attack reaction* phase.

**Firewall technologies** are also used to avoid these attacks, but as chapter 3 shows, it is not the best option. There are two different types of firewalls: application based firewalls and network-based firewalls. The defense mechanisms proposed might be compared with a network-based firewall due to the OSI level where it is implemented. Section 4.1.2.2 explains two different SYN Flooding defense mechanisms based on firewalls. In these cases, both mechanisms aim at using spoofed packets to control the TCP connection process.

The problem of network-based firewalls is that they are not able to access to higher levels in the OSI model, and, as a result, it is not possible to control some aspects like destination and source ports. This aspect is very important in our implementation due to our *Controller* installing *flow entries* with source and destination IP addresses from a server, specifying port 80.

We can conclude saying that separation of the control plane and the data plane in SDN technology provides benefits for each part. With **OpenFlow**, through *Controllers* based on high level programming languages it is possible to orchestrate *OpenFlow Switches* by installing low-level rules on them. This technology abstracts the developer to low-level programming languages too close to the machine. Software developed with OpenFlow have more scalability since there is just one device to control all the network. Since DDoS attacks attack from a big number of different hosts, SDN allows us to control the network topology as a whole.

# Bibliography


[1] ADVISORY CA-1998-01., C. Smurf IP Denial-of-Service attacks. http://www.cert.org/historical/advisories/CA-1998-01.cfm? [Online; accessed 01-Dec-2013].

[2] ARUBANANETWORKS. Configuring Firewall Policies. http://www.arubanetworks.com/techdocs/ArubaOS_63_Web_Help/Content/ArubaFrameStyles/Firewall_Roles/Policies.htm. [Online; accessed 20-June-2014].

[3] BELLOVIN, S. M., LEECH, M., AND TAYLOR, T. Icmp traceback messages. *Internet Engineering Task Force, Marina del Rey, Calif* (2003).

[4] CARL, G., KESIDIS, G., BROOKS, R. R., AND RAI, S. Denial-of-service attack-detection techniques. *Internet Computing, IEEE 10*, 1 (2006), 82–89.

[5] CC., C. Denial of service attack. http://www.cert.org/tech_tips/denial_of_service.html. [Online; accessed 28-Nov-2013].

[6] CHANG, R. K. Defending against flooding-based distributed denial-of-service attacks: A tutorial. *Communications Magazine, IEEE 40*, 10 (2002), 42–51.

[7] EDDY, W. M. Tcp syn flooding attacks and common mitigations. *Verizon Federal Network Systems* (2007).

[8] FERGUSON, P. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. *Amaranth Networks Inc.* (2000).

[9] GUPTA, B., JOSHI, R. C., AND MISRA, M. Defending against distributed denial of service attacks: issues and challenges. *Information Security Journal: A Global Perspective 18*, 5 (2009), 224–247.

[10] Haris, S., Ahmad, R., and Ghani, M. Detecting tcp syn flood attack based on anomaly detection. In *Network Applications Protocols and Services (NETAPPS), 2010 Second INTERNATIONAL Conference on* (2010), IEEE, pp. 240–244.

[11] Hussain, A., Heidemann, J., and Papadopoulos, C. A framework for classifying denial of service attacks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 99–110.

[12] Inc., C. Defenses Against TCP SYN Flooding Attacks. `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-4/syn_flooding_attacks.html`. [Online; accessed 01-Dec-2013].

[13] Keshariya, A., and Foukia, N. Ddos defense mechanisms: a new taxonomy. In *Data Privacy Management and Autonomous Spontaneous Security*. Springer, 2010, pp. 222–236.

[14] Kumarasamy, S., and Gowrishankar, A. An active defense mechanism for tcp syn flooding attacks. *arXiv preprint arXiv:1201.2103* (2012).

[15] Lemon, J., et al. Resisting syn flood dos attacks with a syn cache. In *BSDCon* (2002), vol. 2002, pp. 89–97.

[16] Mahoney, M. V. Network traffic anomaly detection based on packet bytes. In *Proceedings of the 2003 ACM symposium on Applied computing* (2003), ACM, pp. 346–350.

[17] Mininet. Mininet Virtual Network. `http://mininet.org/`. [Online; accessed 28-Jan-2014].

[18] Mirkovic, J., and Reiher, P. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review 34*, 2 (2004), 39–53.

[19] Mölsä, J. Mitigating denial of service attacks: A tutorial. *Journal of computer security 13*, 6 (2005), 807–837.

[20] NOX. POX Controller. `http://www.noxrepo.org/pox/about-pox/`. [Online; accessed 22-Jan-2014].

[21] NOXRepo.org. NOX Controller. `http://www.noxrepo.org/`. [Online; accessed 22-Jan-2014].

[22] OpenFlow. Open Flow standard. `http://archive.openflow.org/`. [Online; accessed 17-Nov-2013].

[23] Paxson, V. Bro: a system for detecting network intruders in real-time. *Computer networks 31*, 23 (1999), 2435–2463.

[24] Peng, T., Leckie, C., and Ramamohanarao, K. Protection from distributed denial of service attacks using history-based ip filtering. In *Communications, 2003. ICC'03. IEEE International Conference on* (2003), vol. 1, IEEE, pp. 482–486.

[25] Peng, T., Leckie, C., and Ramamohanarao, K. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Computing Surveys (CSUR) 39*, 1 (2007), 3.

[26] Radware. Radware DoS and DDoS security report. `http://www.radware.com/assets/0/314/6442478110/6fd67847-ee99-4251-a162-1d09a0e14e48.pdf`. [Online; accessed 18-May-2014].

[27] Radware. Radware main page. `http://www.radware.com/`. [Online; accessed 18-May-2014].

[28] Roesch, M., et al. Snort: Lightweight intrusion detection for networks. In *LISA* (1999), vol. 99, pp. 229–238.

[29] Schuba, C. L., Krsul, I. V., Kuhn, M. G., Spafford, E. H., Sundaram, A., and Zamboni, D. Analysis of a denial of service attack on tcp. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on* (1997), IEEE, pp. 208–223.

[30] sFlow. sFlow Blog and API. `http://blog.sflow.com/2011/01/presentation.html`. [Online; accessed 16-Mar-2014].

[31] sFlow. sFlow Monitoring Networks. `http://www.sflow.org/`. [Online; accessed 14-Mar-2014].

[32] Siris, V. A., and Papagalou, F. Application of anomaly detection algorithms for detecting syn flooding attacks. *Computer communications 29*, 9 (2006), 1433–1442.

[33] Specht, S. M., and Lee, R. B. Distributed denial of service: Taxonomies of attacks, tools, and countermeasures. In *ISCA PDCS* (2004), pp. 543–550.

[34] Specification, O. S. Version 1.4.0. *Open Networking Foundation* (2013).

[35] Srivastava, A., Gupta, B., Tyagi, A., Sharma, A., and Mishra, A. A recent survey on ddos attacks and defense mechanisms. In *Advances in Parallel Distributed Computing*. Springer, 2011, pp. 570–580.

[36] Stallings, W. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987.

[37] Yau, D. K., Lui, J., Liang, F., and Yam, Y. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Transactions on Networking (TON) 13*, 1 (2005), 29–42.

# Appendix A

# Protocol Headers Structure

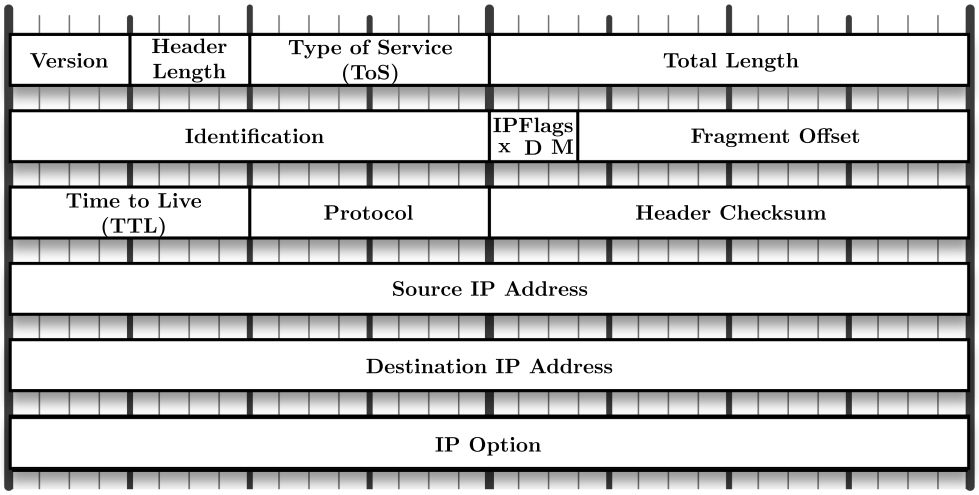



Figure A.1: IP V.4 Header

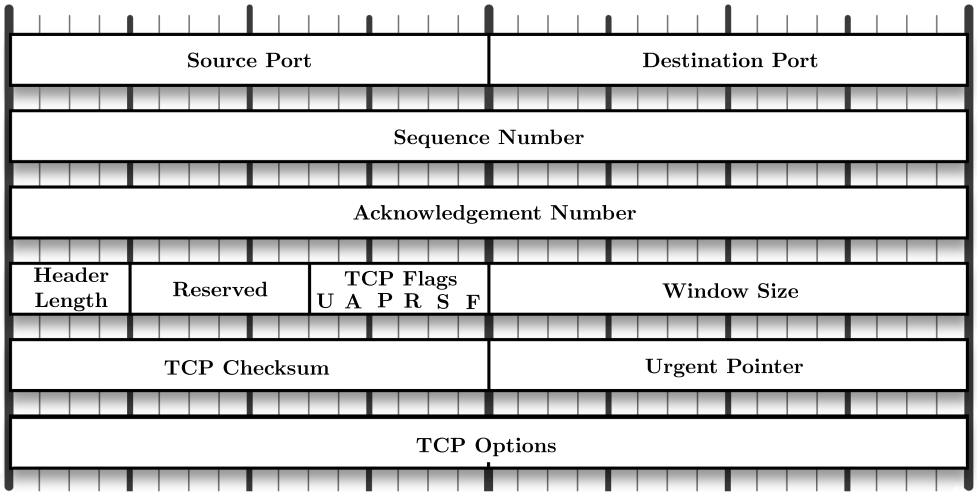



Figure A.2: TCP Header

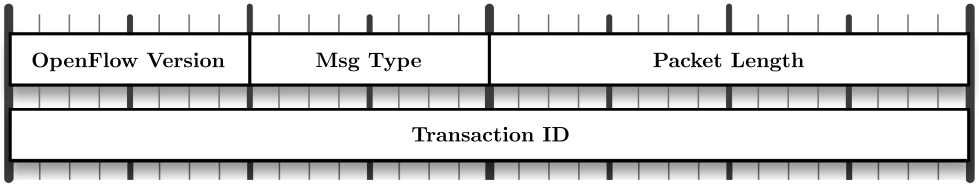



Figure A.3: OpenFlow Protocol Header

# Appendix B

# Result captions

## B.1 Terminal Host

```
--2014-06-17 16:13:42--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1264 (1.2K) [text/html]
Saving to: `index.html.521'

100%[======================================>] 1,264       --.-K/s   in 0s

2014-06-17 16:13:42 (12.4 MB/s) - `index.html.521' saved [1264/1264]
```

(a) Host behaviour before attack

```
--2014-06-17 16:21:10--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1264 (1.2K) [text/html]
Saving to: `index.html.608.1'

100%[============================>] 1,264       --.-K/s   in 0s

2014-06-17 16:21:32 (19.3 MB/s) - `index.html.608.1' saved [1264/1264]

--2014-06-17 16:22:02--  http://10.0.0.1/
Connecting to 10.0.0.1:80... failed: No route to host.
--2014-06-17 16:22:50--  http://10.0.0.1/
Connecting to 10.0.0.1:80... failed: No route to host.
--2014-06-17 16:23:24--  http://10.0.0.1/
Connecting to 10.0.0.1:80...
```

(b) Host behaviour during attack

Figure B.1: Terminal Host without a Defense Mechanism

```
--2014-06-17 16:59:51--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1264 (1.2K) [text/html]
Saving to: `index.html.633'

100%[======================================>] 1,264       --.-K/s   in 0s

2014-06-17 16:59:51 (43.8 MB/s) - `index.html.633' saved [1264/1264]

```

(a) Host behaviour before attack

```
--2014-06-17 17:00:32--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1264 (1.2K) [text/html]
Saving to: `index.html.644'

100%[======================================>] 1,264       --.-K/s   in 0s

2014-06-17 17:00:32 (42.2 MB/s) - `index.html.644' saved [1264/1264]

```

(b) Host behaviour during attack

Figure B.2: Terminal Host with a Defense Mechanism

## B.2 OpenFlow Table

```
ninet@mininet-vm:~$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x7fa25629): flags=none type=1(flow)

  cookie=0, duration_sec=10s, duration_nsec=435000000s,
  table_id=0, priority=32768, n_packets=5, n_bytes=1788,
  idle_timeout=60, hard_timeout=0,ip,nw_src=10.0.0.1,
  nw_dst=10.0.0.6,actions=output:6

  cookie=0, duration_sec=10s, duration_nsec=435000000s,
  table_id=0, priority=32768, n_packets=5, n_bytes=438,
  idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.6,
  nw_dst=10.0.0.1,actions=output:1

  cookie=0, duration_sec=31s, duration_nsec=528000000s,
  table_id=0, priority=32768, n_packets=11, n_bytes=834,
  idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.5,
```

```
nw_dst=10.0.0.1,actions=output:1

cookie=0, duration_sec=192s, duration_nsec=359000000s,
table_id=0, priority=32768, n_packets=79, n_bytes=15412,
idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.1,
nw_dst=10.0.0.4,actions=output:4

cookie=0, duration_sec=192s, duration_nsec=991000000s,
table_id=0, priority=32768, n_packets=456, n_bytes=34504,
idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.3,
nw_dst=10.0.0.1,actions=output:1

cookie=0, duration_sec=192s, duration_nsec=396000000s,
table_id=0, priority=32768, n_packets=77, n_bytes=5778,
idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.4,
nw_dst=10.0.0.1,actions=output:1

cookie=0, duration_sec=31s, duration_nsec=526000000s,
table_id=0, priority=32768, n_packets=12, n_bytes=2250,
idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.1,
nw_dst=10.0.0.5,actions=output:5

cookie=0, duration_sec=192s, duration_nsec=954000000s,
table_id=0, priority=32768, n_packets=452, n_bytes=86686,
idle_timeout=60,hard_timeout=0,ip,nw_src=10.0.0.1,
nw_dst=10.0.0.3,actions=output:3
```